Chipsmall Limited consists of a professional team with an average of over 10 year of expertise in the distribution of electronic components. Based in Hongkong, we have already established firm and mutual-benefit business relationships with customers from,Europe,America and south Asia,supplying obsolete and hard-to-find components to meet their specific needs.

With the principle of “Quality Parts,Customers Priority,Honest Operation,and Considerate Service”,our business mainly focus on the distribution of electronic components. Line cards we deal with include Microchip,ALPS,ROHM,Xilinx,Pulse,ON,Everlight and Freescale. Main products comprise IC,Modules,Potentiometer,IC Socket,Relay,Connector.Our parts cover such applications as commercial,industrial, and automotives areas.

We are looking forward to setting up business relationship with you and hope to provide you with the best service and solution. Let us make a better world for our industry!

Contact us

Tel: +86-755-8981 8866 Fax: +86-755-8427 6832
Email & Skype: info@chipsmall.com Web: www.chipsmall.com
Address: A1208, Overseas Decoration Building, #122 Zhenhua RD., Futian, Shenzhen, China

PM0215
Programming manual

STM32F0xxx Cortex-M0 programming manual

Introduction

This programming manual provides information for application and system-level software developers. It gives a full description of the STM32 Cortex™-M0 processor programming model, instruction set and core peripherals.

The STM32 Cortex™-M0 processor is a high performance 32-bit processor designed for the microcontroller market. It offers significant benefits to developers, including:

- Outstanding processing performance combined with fast interrupt handling
- Enhanced system debug with extensive breakpoint and trace capabilities
- Efficient processor core, system and memories
- Ultra-low power consumption with integrated sleep modes
- Platform security

Table 1. Applicable products

Type	Part numbers
Microcontroller	STM32F0xxx

Contents

1 About this document 8

1.1 Typographical conventions 8

1.2 List of abbreviations for registers 8

1.3 About the STM32 Cortex-M0 processor and core peripherals 9

1.3.1 System level interface 10

1.3.2 Integrated configurable debug 10

1.3.3 Cortex-M0 processor features and benefits summary 10

1.3.4 Cortex-M0 core peripherals 10

2 The STM32 Cortex-M0 processor 11

2.1 Programmers model 11

2.1.1 Processor modes 11

2.1.2 Stacks 11

2.1.3 Core registers 12

2.1.4 Exceptions and interrupts 17

2.1.5 Data types 17

2.1.6 The Cortex microcontroller software interface standard (CMSIS) 17

2.2 Memory model 18

2.2.1 Memory regions, types and attributes 19

2.2.2 Memory system ordering of memory accesses 19

2.2.3 Behavior of memory accesses 20

2.2.4 Software ordering of memory accesses 20

2.2.5 Memory endianness 21

2.3 Exception model 22

2.3.1 Exception states 22

2.3.2 Exception types 22

2.3.3 Exception handlers 23

2.3.4 Vector table 24

2.3.5 Exception priorities 25

2.3.6 Exception entry and return 25

2.4 Fault handling 28

2.5 Power management 28

2.5.1 Entering sleep mode 29

2.5.2 Wakeup from sleep mode 29
2.5.3 The external event input 30
2.5.4 Power management programming hints 30

3 The STM32 Cortex-M0 instruction set 31

3.1 Instruction set summary 31
3.2 CMSIS intrinsic functions 35
3.3 About the instruction descriptions 36
3.3.1 Operands 36
3.3.2 Restrictions when using PC or SP 36
3.3.3 Shift operations 36
3.3.4 Address alignment 39
3.3.5 PC-relative expressions 39
3.3.6 Conditional execution 39
3.4 Memory access instructions 41
3.4.1 ADR 42
3.4.2 LDR and STR, immediate offset 43
3.4.3 LDR and STR, register offset 44
3.4.4 LDR, PC-relative 45
3.4.5 LDM and STM 46
3.4.6 PUSH and POP 47
3.5 General data processing instructions 48
3.5.1 ADD{S}, ADCS, SUB{S}, SBCS, and RSBS 49
3.5.2 ANDS, ORRS, EORS and BICS 51
3.5.3 ASRS, LSLS, LSRS and RORS 52
3.5.4 CMP and CMN 53
3.5.5 MOV, MOVS and MVNS 54
3.5.6 MULS 55
3.5.7 REV, REV16, and REVSH 56
3.5.8 SXTB, SXTH, UXTB and UXTH 57
3.5.9 TST 58
3.6 Branch and control instructions 59
3.6.1 B, BL, BX, and BLX 59
3.7 Miscellaneous instructions 61
3.7.1 BKPT 61
3.7.2 CPSID CPSIE 62

3.7.3 DMB 63
3.7.4 DSB 63
3.7.5 ISB 64
3.7.6 MRS 64
3.7.7 MSR 65
3.7.8 NOP 66
3.7.9 SEV 66
3.7.10 SVC 67
3.7.11 WFE 67
3.7.12 WFI 68

4 Core peripherals 69

4.1 About the STM32 Cortex-M0 core peripherals 69
4.2 Nested vectored interrupt controller (NVIC) 70
4.2.1 Accessing the Cortex-M0 NVIC registers using CMSIS 70
4.2.2 Interrupt set-enable register (ISER) 71
4.2.3 Interrupt clear-enable register (ICER) 71
4.2.4 Interrupt set-pending register (ISPR) 72
4.2.5 Interrupt clear-pending register (ICPR) 72
4.2.6 Interrupt priority register (IPR0-IPR7) 73
4.2.7 Level-sensitive and pulse interrupts 74
4.2.8 NVIC design hints and tips 75
4.2.9 NVIC register map 76
4.3 System control block (SCB) 77
4.3.1 CPUID base register (CPUID) 77
4.3.2 Interrupt control and state register (ICSR) 78
4.3.3 Application interrupt and reset control register (AIRCR) 80
4.3.4 System control register (SCR) 81
4.3.5 Configuration and control register (CCR) 82
4.3.6 System handler priority registers (SHPRx) 83
4.3.7 SCB usage hints and tips 84
4.3.8 SCB register map 84
4.4 SysTick timer (STK) 85
4.4.1 SysTick control and status register (STK_CSR) 86
4.4.2 SysTick reload value register (STK_RVR) 87
4.4.3 SysTick current value register (STK_CVR) 87
4.4.4 SysTick calibration value register (STK_CALIB) 88

4.4.5 SysTick design hints and tips 88
4.4.6 SysTick register map 89

5 Revision history 90

List of tables

Table 1. Applicable products 1
Table 2. Summary of processor mode and stack usage 11
Table 3. Core register set summary 12
Table 4. PSR register combinations and attributes 13
Table 5. APSR bit definitions 14
Table 6. IPSR bit definitions 14
Table 7. EPSR bit definitions 15
Table 8. PRIMASK register bit definitions. 16
Table 9. CONTROL register bit definitions 16
Table 10. Ordering of memory accesses 19
Table 11. Memory access behavior 20
Table 12. Properties of the different exception types 23
Table 13. Exception return behavior. 27
Table 14. Cortex-M0 instructions 31
Table 15. CMSIS intrinsic functions to generate some Cortex-M0 instructions 35
Table 16. CMSIS intrinsic functions to access the special registers. 35
Table 17. Condition code suffixes and their relationship with the flags 40
Table 18. Memory access instructions 41
Table 19. Data processing instructions. 48
Table 20. ADCS, ADD, RSBS, SBCS and SUB operand restrictions. 50
Table 21. Branch and control instructions 59
Table 22. Branch ranges 59
Table 23. Miscellaneous instructions 61
Table 24. STM32 core peripheral register regions 69
Table 25. NVIC register summary 70
Table 26. CMSIS access NVIC functions 70
Table 27. IPR bit assignments 73
Table 28. CMSIS functions for NVIC control 75
Table 29. NVIC register map and reset values. 76
Table 30. Summary of the system control block registers 77
Table 31. System fault handler priority fields and registers 83
Table 32. SCB register map and reset values 84
Table 33. System timer registers summary 85
Table 34. SysTick register map and reset values. 89
Table 35. Document revision history 90

List of figures

Figure 1. STM32 Cortex-M0 implementation 9
Figure 2. Processor core registers 12
Figure 3. APSR, IPSR and EPSR bit assignments 13
Figure 4. PRIMASK register bit assignments 15
Figure 5. CONTROL register bit assignments 16
Figure 6. Memory map 18
Figure 7. Little-endian example 21
Figure 8. Vector table 24
Figure 9. Cortex-M0 stack frame layout 26
Figure 10. ASR#3 37
Figure 11. LSR#3 37
Figure 12. LSL#3 38
Figure 13. ROR #3 38
Figure 14. IPR register mapping 73

1 About this document

This document provides the information required for application and system-level software development. It does not provide information on debug components, features, or operation.

This material is for microcontroller software and hardware engineers, including those who have no experience of ARM products.

1.1 Typographical conventions

The typographical conventions used in this document are:

italic	Highlights important notes, introduces special terminology, denotes internal cross-references, and citations.
< and >	Enclose replaceable terms for assembler syntax where they appear in code or code fragments. For example: LDRSB<cond> <Rt>, [<Rn>, #<offset>]
bold	Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.
`monospace`	Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.
`monospace` (with "mono" underlined)	Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
`monospace italic`	Denotes arguments to monospace text where the argument is to be replaced by a specific value.
`monospace bold`	Denotes language keywords when used outside example code.

1.2 List of abbreviations for registers

The following abbreviations are used in register descriptions:

read/write (rw)	Software can read and write to these bits.
read-only (r)	Software can only read these bits.
write-only (w)	Software can only write to this bit. Reading the bit returns the reset value.
read/clear (rc_w1)	Software can read as well as clear this bit by writing 1. Writing ‘0’ has no effect on the bit value.
read/clear (rc_w0)	Software can read as well as clear this bit by writing 0. Writing ‘1’ has no effect on the bit value.
toggle (t)	Software can only toggle this bit by writing ‘1’. Writing ‘0’ has no effect.
Reserved (Res.)	Reserved bit, must be kept at reset value.

1.3 About the STM32 Cortex-M0 processor and core peripherals

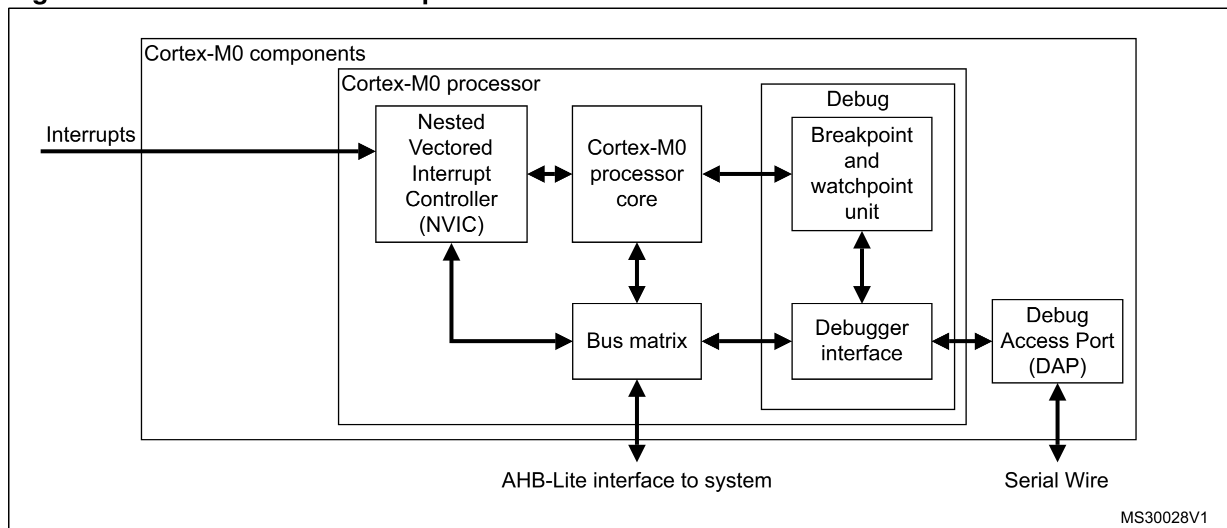
The Cortex-M0 processor is an entry-level 32-bit ARM Cortex processor designed for a broad range of embedded applications. It offers significant benefits to developers, including:

- a simple architecture that is easy to learn and program
- ultra-low power, energy efficient operation
- excellent code density
- deterministic, high-performance interrupt handling
- upward compatibility with Cortex-M processor family.

The Cortex-M0 processor is built on a highly area and power optimized 32-bit processor core, with a 3-stage pipeline von Neumann architecture. The processor delivers exceptional energy efficiency through a small but powerful instruction set and extensively optimized design, providing high-end processing hardware including a single-cycle multiplier.

The Cortex-M0 processor implements the ARMv6-M architecture, which is based on the 16-bit Thumb® instruction set and includes Thumb-2 technology. This provides the exceptional performance expected of a modern 32-bit architecture, with a higher code density than other 8-bit and 16-bit microcontrollers.

Figure 1. STM32 Cortex-M0 implementation



The Cortex-M0 processor closely integrates a configurable nested vectored interrupt controller (NVIC), to deliver industry-leading interrupt performance. The NVIC:

- includes a non-maskable interrupt (NMI)
- provides zero jitter interrupt option
- provides four interrupt priority levels.

The tight integration of the processor core and NVIC provides fast execution of interrupt service routines (ISRs), dramatically reducing the interrupt latency. This is achieved through the hardware stacking of registers, and the ability to abandon and restart load-multiple and store-multiple operations. Interrupt handlers do not require any assembler wrapper code, removing any code overhead from the ISRs. Tail-chaining optimization also significantly reduces the overhead when switching from one ISR to another. To optimize low-power designs, the NVIC integrates with the sleep modes, including a deep sleep function that enables the entire device to be rapidly powered down.

1.3.1 System level interface

The Cortex-M0 processor provides a single system-level interface using AMBA® technology to provide high speed, low latency memory accesses.

1.3.2 Integrated configurable debug

The Cortex-M0 processor implements a complete hardware debug solution, with extensive hardware breakpoint and watchpoint options. This provides high system visibility of the processor, memory and peripherals through a 2-pin Serial Wire Debug (SWD) port that is ideal for small package devices.

1.3.3 Cortex-M0 processor features and benefits summary

- High code density with 32-bit performance
- Tools and binary upwards compatible with Cortex-M processor family
- Integrated ultra low-power sleep modes
- Efficient code execution permits slower processor clock or increases sleep mode time
- Single-cycle 32-bit hardware multiplier
- Zero jitter interrupt handling
- Extensive debug capabilities

1.3.4 Cortex-M0 core peripherals

The peripherals are:

- Nested vectored interrupt controller: The NVIC is an embedded interrupt controller that supports low latency interrupt processing.
- System control block: The SCB is the programmers model interface to the processor. It provides system implementation information and system control, including configuration, control, and reporting of system exceptions.
- System timer: SysTick is a 24-bit count-down timer. Use this as a Real Time Operating System (RTOS) tick timer or as a simple counter.

2 The STM32 Cortex-M0 processor

2.1 Programmers model

This section describes the Cortex-M0 programmers model. In addition to the individual core register descriptions, it contains information about the processor modes and stacks.

2.1.1 Processor modes

The processor modes are:

Thread mode: Used to execute application software.
The processor enters Thread mode when it comes out of reset.

Handler mode: Used to handle exceptions.
The processor returns to Thread mode when it has finished exception processing.

The Cortex-M0 does not support multiple privilege levels. It can always use all instructions and access all resources.

2.1.2 Stacks

The processor uses a full descending stack. This means the stack pointer indicates the last stacked item on the stack memory. When the processor pushes a new item onto the stack, it decrements the stack pointer and then writes the item to the new memory location.

The processor implements two stacks, with independent copies of the stack pointer,(see *Stack pointer (SP) register R13 on page 13*):

- the main stack and
- the process *stack*,

In Thread mode, the CONTROL register controls whether the processor uses the main stack or the process stack, see *Control register on page 16*.

In Handler mode, the processor always uses the main stack.

The options for processor operations are:

Table 2. Summary of processor mode and stack usage

Processor mode	Used to execute	Stack used
Thread	Applications	Main stack or process stack [1]
Handler	Exception handlers	Main stack

1. See *Control register on page 16*.

2.1.3 Core registers

Figure 2. Processor core registers

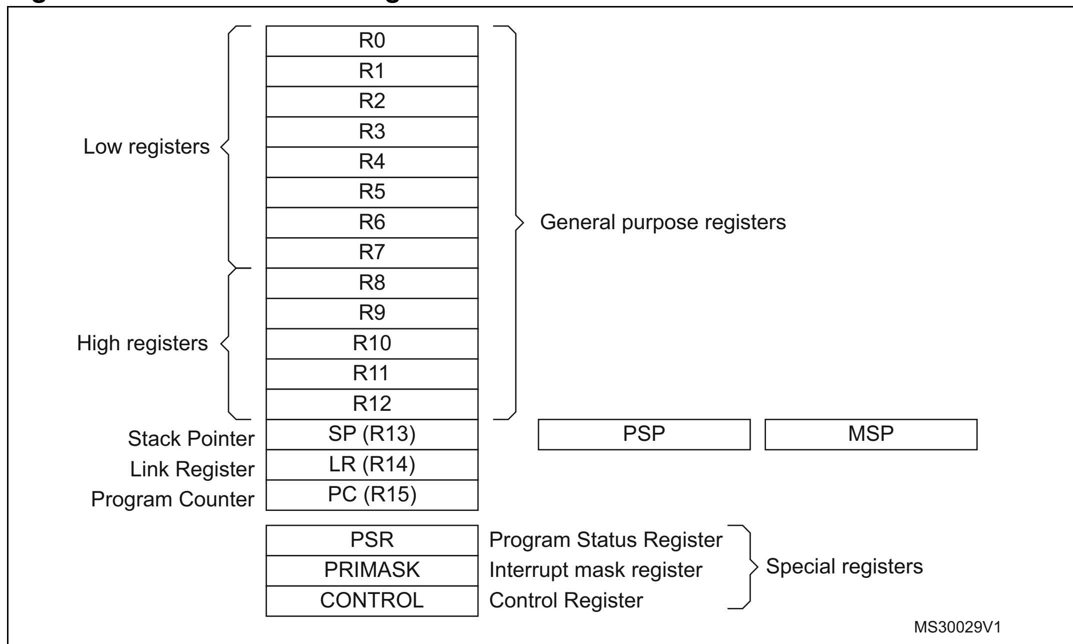


Table 3. Core register set summary

Name	Type(1)	Reset value	Description
R0-R12	read-write	Unknown	*General-purpose registers on page 12*
MSP	read-write	See description	*Stack pointer (SP) register R13 on page 13*
PSP	read-write	Unknown	*Stack pointer (SP) register R13 on page 13*
LR	read-write	Unknown	*Link register (LR) register R14 on page 13*
PC	read-write	See description	*Program counter (PC) register R15 on page 13*
PSR	read-write	Unknown(2)	*Program status register on page 13*
ASPR	read-write	Unknown	*Application program status register on page 14*
IPSR	read-only	0x00000000	*Interrupt program status register on page 14*
EPSR	read-only	Unknown(2)	*Execution program status register on page 15*
PRIMASK	read-write	0x00000000	*Priority mask register on page 15*
CONTROL	read-write	0x00000000	*Control register on page 16*

1. Describes access type during program execution in Thread and Handler modes. Debug access can differ.
2. Bit[24] is the T-bit and is loaded from bit[0] of the reset vector.

General-purpose registers

R0-R12 are 32-bit general-purpose registers for data operations.

Stack pointer (SP) register R13

In Thread mode, bit[1] of the CONTROL register indicates the stack pointer to use:

- 0: Main Stack Pointer (MSP)(reset value). On reset, the processor loads the MSP with the value from address 0x00000000.
- 1: Process Stack Pointer (PSP).

Link register (LR) register R14

Stores return information for subroutines, function calls, and exceptions. On reset, the processor loads the LR value 0xFFFFFFFF.

Program counter (PC) register R15

Contains the current program address. On reset, the processor loads the PC with the value of the reset vector, which is at address 0x00000004. Bit[0] of the value is loaded into the EPSR T-bit at reset and must be 1.

Program status register

The *Program Status Register* (PSR) combines:

- *Application program status register* (APSR)
- *Interrupt program status register* (IPSR)
- *Execution program status register* (EPSR)

These registers are mutually exclusive bitfields in the 32-bit PSR. They can be accessed individually or as a combination of any two or all three registers, using the register name as an argument to the *MSR* or *MRS* instructions. For example:

- Read all of the registers using PSR with the MRS instruction
- Write to the APSR using APSR with the MSR instruction.

Figure 3. APSR, IPSR and EPSR bit assignments

	31	30	29	28	27 ... 25	24	23 ... 6	5 ... 0
APSR	N	Z	C	V	Reserved			
IPSR	Reserved							Exception number
EPSR	Reserved					T	Reserved	

Table 4. PSR register combinations and attributes

Register	Type	Combination
PSR	read-write(1), (2)	APSR, EPSR, and IPSR
IEPSR	read-only	EPSR and IPSR
IAPSR	read-write(1)	APSR and IPSR
EAPSR	read-write(2)	APSR and EPSR

1. The processor ignores writes to the IPSR bits.
2. Reads of the EPSR bits return zero, and the processor ignores writes to the these bits

Application program status register

Contains the current state of *The condition flags* from previous instruction executions. See the register summary in *Table 3 on page 12* for its attributes.

Table 5. APSR bit definitions

Bits	Description
Bit 31	**N:** Negative or less than flag: 0: Operation result was positive, zero, greater than, or equal 1: Operation result was negative or less than.
Bit 30	**Z:** Zero flag: 0: Operation result was not zero 1: Operation result was zero.
Bit 29	**C:** Carry or borrow flag: 0: Add operation did not result in a carry bit or subtract operation resulted in a borrow bit 1: Add operation resulted in a carry bit or subtract operation did not result in a borrow bit.
Bit 28	**V:** Overflow flag: 0: Operation did not result in an overflow 1: Operation resulted in an overflow.
Bits 27:0	Reserved.

Interrupt program status register

Contains the exception type number of the current *Interrupt Service Routine* (ISR). See the register summary in *Table 3 on page 12* for its attributes.

Table 6. IPSR bit definitions

Bits	Description
Bits 31:6	Reserved
Bits 5:0	**ISR_NUMBER**: This is the number of the current exception, see *Exception types on page 22* for more information: 0: Thread mode 1: Resetrved 2: NMI 3: Hard fault 4-10: Reserved 11: SVCall 12: Reserved 13: Reserved 14: PendSV 15: SysTick/Reserved 16: IRQ0 .... 47: IRQ31 (see STM32 product reference manual/datasheet for interrupt mapping information) 48-63: Reserved

Execution program status register

The EPSR contains the Thumb state bit.

See the register summary in *Table 3 on page 12* for the EPSR attributes. The bit assignments are:

Table 7. EPSR bit definitions

Bits	Description
Bits 31:25	Reserved.
Bit 24	T: Thumb state bit.
Bits 23:0	Reserved.

Attempts to read the EPSR directly through application software using the MSR instruction always return zero. Attempts to write the EPSR using the MSR instruction in application software are ignored. Fault handlers can examine EPSR value in the stacked PSR to indicate the operation that is at fault. See *Section 2.3.6: Exception entry and return on page 25*.

The following can clear the T bit to 0:

- instructions BLX, BX and POP{PC}
- restoration from the stacked xPSR value on an exception return
- bit[0] of the vector value on an exception entry.

Attempting to execute instructions when the T bit is 0 results in a HardFault or lockup. See *Lockup on page 28* for more information.

Interruptable-restartable instructions

LDM and STM are interruptable-restartable instructions. If an interrupt occurs during the execution of one of these instructions, the processor abandons execution of the instruction. After servicing the interrupt, the processor restarts execution of the instruction from the beginning.

Exception mask registers

The exception mask registers disable the handling of exceptions by the processor. Disable exceptions where they might impact on timing critical tasks or code sequences.

To disable or re-enable exceptions use the MSR and MRS instructions, or the CPS instruction to change the value of PRIMASK. See *MRS on page 64*, *MSR on page 65*, and *CPSID CPSIE on page 62* for more information.

Priority mask register

The PRIMASK register prevents activation of all exceptions with configurable priority. See the register summary in *Table 3 on page 12* for its attributes.

Figure 4. PRIMASK register bit assignments

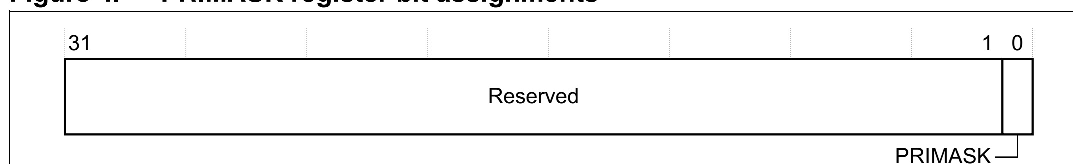


Table 8. PRIMASK register bit definitions

Bits	Description
Bits 31:1	Reserved
Bit 0	**PRIMASK**: 0: No effect 1: Prevents the activation of all exceptions with configurable priority.

Control register

The CONTROL register controls the stack used when the processor is in Thread mode. See the register summary in *Table 3 on page 12* for its attributes.

Figure 5. CONTROL register bit assignments

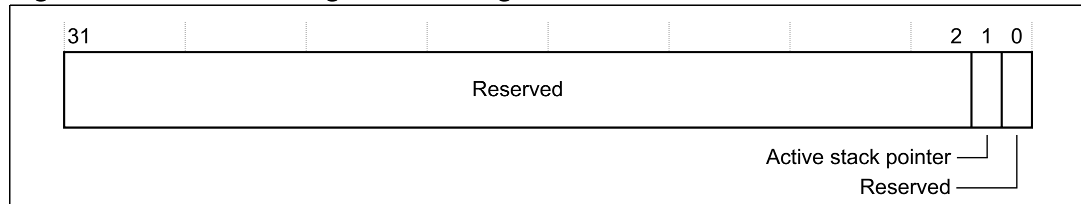


Table 9. CONTROL register bit definitions

Bits	Function
Bits 31:2	Reserved
Bit 1	**ASPSEL**: Active stack pointer selection. Selects the current stack: 0: MSP is the current stack pointer 1: PSP is the current stack pointer. In Handler mode this bit reads as zero and ignores writes.
Bit 0	Reserved

Handler mode always uses the MSP, so the processor ignores explicit writes to the active stack pointer bit of the CONTROL register when in Handler mode. The exception entry and return mechanisms update the CONTROL register.

In an OS environment, it is recommended that threads running in Thread mode use the process stack and the kernel and exception handlers use the main stack. By default, Thread mode uses the MSP. To switch the stack pointer used in Thread mode to the PSP, use the MSR instruction to set the Active stack pointer bit to 1, see *MSR on page 65*. When changing the stack pointer, software must use an ISB instruction immediately after the MSR instruction. This ensures that instructions after the ISB execute using the new stack pointer. See *ISB on page 64*

2.1.4 Exceptions and interrupts

The Cortex-M0 processor supports interrupts and system exceptions. The processor and the NVIC prioritize and handle all exceptions. An exception changes the normal flow of software control. The processor uses handler mode to handle all exceptions except for reset. See *Exception entry on page 26* and *Exception return on page 27* for more information. The NVIC registers control interrupt handling. See *Nested vectored interrupt controller (NVIC) on page 70* for more information.

2.1.5 Data types

The processor manages all memory accesses as little-endian. See *Memory regions, types and attributes on page 19* for more information. It supports the following data types:

- 32-bit words
- 16-bit halfwords
- 8-bit bytes

2.1.6 The Cortex microcontroller software interface standard (CMSIS)

ARM provides the Cortex Microcontroller Software Interface Standard (CMSIS) for programming Cortex-M0 microcontrollers. The CMSIS is an integrated part of the device driver library. For a Cortex-M0 microcontroller system, the *Cortex Microcontroller Software Interface Standard* (CMSIS) defines:

- A common way to:
  - Access peripheral registers
  - Define exception vectors
- The names of:
  - The registers of the core peripherals
  - The core exception vectors
- A device-independent interface for RTOS kernels.

The CMSIS includes address definitions and data structures for the core peripherals in the Cortex-M0 processor.

The CMSIS simplifies software development by enabling the reuse of template code and the combination of CMSIS-compliant software components from various middleware vendors. Software vendors can expand the CMSIS to include their peripheral definitions and access functions for those peripherals.

This document includes the register names defined by the CMSIS, and gives short descriptions of the CMSIS functions that address the processor core and the core peripherals.

Note: *This document uses the register short names defined by the CMSIS. In a few cases these differ from the architectural short names that might be used in other documents.*

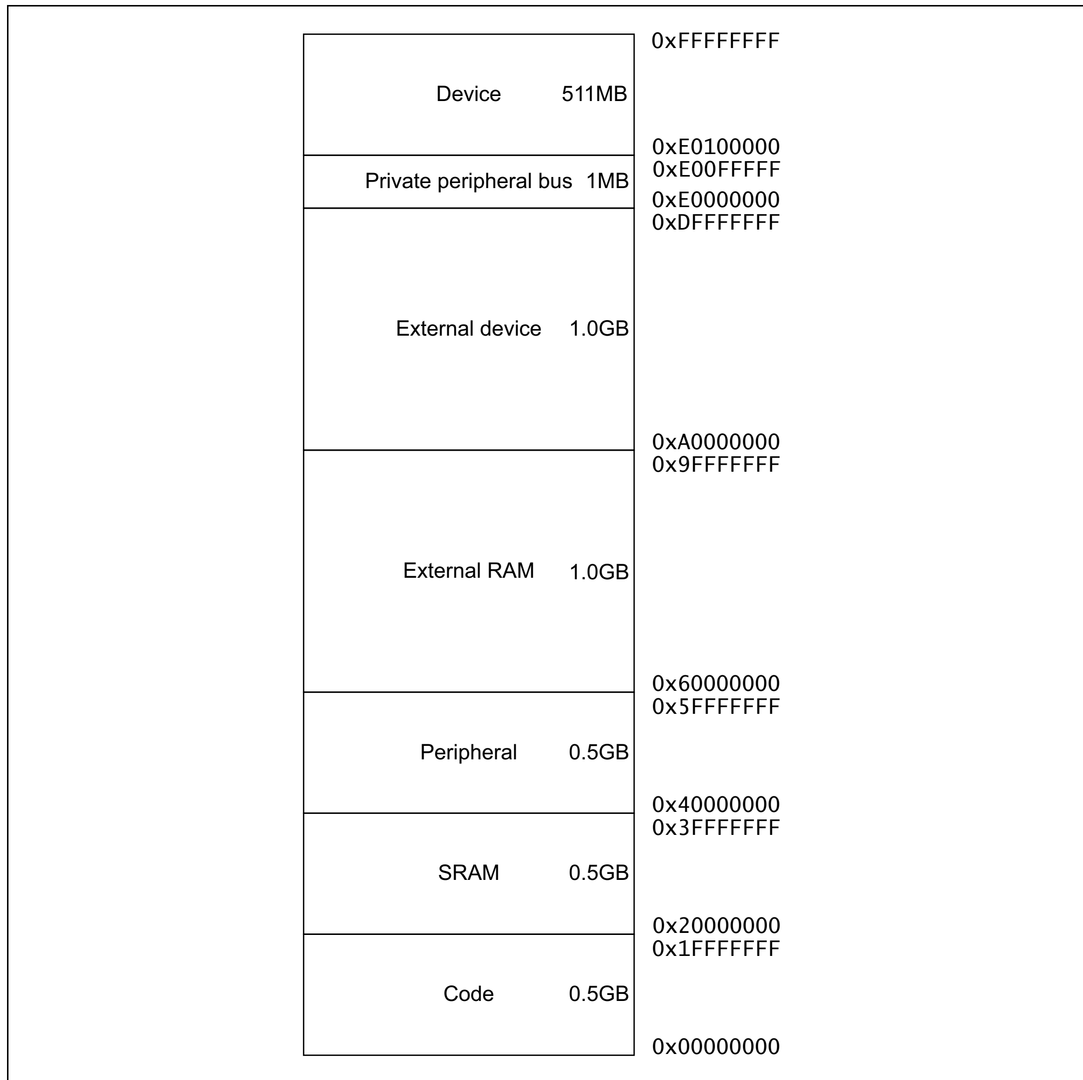
The following sections give more information about the CMSIS:

- *Power management programming hints on page 30*
- *CMSIS intrinsic functions on page 35*
- *Interrupt set-enable register (ISER) on page 71*
- *NVIC programming hints on page 75*

2.2 Memory model

This section describes the processor memory map, and the behavior of memory accesses. The processor has a fixed memory map that provides up to 4 GB of addressable memory.

Figure 6. Memory map



The processor reserves regions of the *Private peripheral bus* (PPB) address range for core peripheral registers, see *Section 4.1: About the STM32 Cortex-M0 core peripherals on page 69*.

2.2.1 Memory regions, types and attributes

The memory map is split into regions. Each region has a defined memory type, and some regions have additional memory attributes. The memory type and attributes determine the behavior of accesses to the region.

The memory types are:

Normal — The processor can re-order transactions for efficiency, or perform speculative reads.

Device — The processor preserves transaction order relative to other transactions to Device or Strongly-ordered memory.

Strongly-ordered — The processor preserves transaction order relative to all other transactions.

The different ordering requirements for Device and Strongly-ordered memory mean that the memory system can buffer a write to Device memory, but must not buffer a write to Strongly-ordered memory.

Additional memory attributes include:

Execute Never (XN) — Means the processor prevents instruction accesses. Any attempt to fetch an instruction from an XN region causes a HardFault exception.

2.2.2 Memory system ordering of memory accesses

For most memory accesses caused by explicit memory access instructions, the memory system does not guarantee that the order in which the accesses complete matches the program order of the instructions, providing this does not affect the behavior of the instruction sequence. Normally, if correct program execution depends on two memory accesses completing in program order, software must insert a memory barrier instruction between the memory access instructions, see *Section 2.2.4: Software ordering of memory accesses on page 20*.

However, the memory system does guarantee some ordering of accesses to Device and Strongly-ordered memory. For two memory access instructions A1 and A2, if A1 occurs before A2 in program order, the ordering of the memory accesses caused by two instructions is:

Table 10. Ordering of memory accesses(1)

A1	A2			
	Normal access	Device access		Strongly ordered access
		Non-shareable	Shareable	
Normal access	-	-	-	-
Device access, non-shareable	-	<	-	<
Device access, shareable	-	-	<	<
Strongly ordered access	-	<	<	<

1. - means that the memory system does not guarantee the ordering of the accesses.
< means that accesses are observed in program order, that is, A1 is always observed before A2.

2.2.3 Behavior of memory accesses

The behavior of accesses to each region in the memory map is:

Table 11. Memory access behavior

Address range	Memory region	Memory type[1]	XN[1]	Description
0x00000000-0x1FFFFFFF	Code	Normal	-	Executable region for program code. Can also put data here.
0x20000000-0x3FFFFFFF	SRAM	Normal	-	Executable region for data. Can also put code here.
0x40000000-0x5FFFFFFF	Peripheral	Device	XN	External device memory
0x60000000-0x9FFFFFFF	External RAM	Normal	-	Executable region for data.
0xA0000000-0xDFFFFFFF	External device	Device	XN	External device memory
0xED000000-0xED0FFFFF	Private Peripheral Bus	Strongly-ordered	XN	This region includes the NVIC, System timer, and system control block. Only word accesses can be used in this region.
0xED100000-0xFFFFFFFF	Device	Device	XN	This region includes all the STM32 standard peripherals.

1. See *Memory regions, types and attributes on page 19* for more information.

The Code, SRAM, and external RAM regions can hold programs.

2.2.4 Software ordering of memory accesses

The order of instructions in the program flow does not always guarantee the order of the corresponding memory transactions. This is because:

- The processor can reorder some memory accesses to improve efficiency, providing this does not affect the behavior of the instruction sequence.
- Memory or devices in the memory map have different wait states
- Some memory accesses are buffered or speculative.

Section 2.2.2: Memory system ordering of memory accesses on page 19 describes the cases where the memory system guarantees the order of memory accesses. Otherwise, if the order of memory accesses is critical, software must include memory barrier instructions to force that ordering. The processor provides the following memory barrier instructions:

DMB The *Data Memory Barrier* instruction ensures that outstanding memory transactions complete before subsequent memory transactions. See *DMB on page 63*.

DSB The *Data Synchronization Barrier* instruction ensures that outstanding memory transactions complete before subsequent instructions execute. See *DSB on page 63*.

ISB The *Instruction Synchronization Barrier* ensures that the effect of all completed memory transactions is recognizable by subsequent instructions. See *ISB on page 64*.

Use memory barrier instructions in, for example:

- **Vector table**: If the program changes an entry in the vector table, and then enables the corresponding exception, use a DMB instruction between the operations. This ensures that if the exception is taken immediately after being enabled the processor uses the new exception vector.
- **Self-modifying code**: If a program contains self-modifying code, use an ISB instruction immediately after the code modification in the program. This ensures subsequent instruction execution uses the updated program.
- **Memory map switching**: If the system contains a memory map switching mechanism, use a DSB instruction after switching the memory map in the program. This ensures subsequent instruction execution uses the updated memory map.

Memory accesses to Strongly-ordered memory, such as the system control block, do not require the use of DMB instructions.

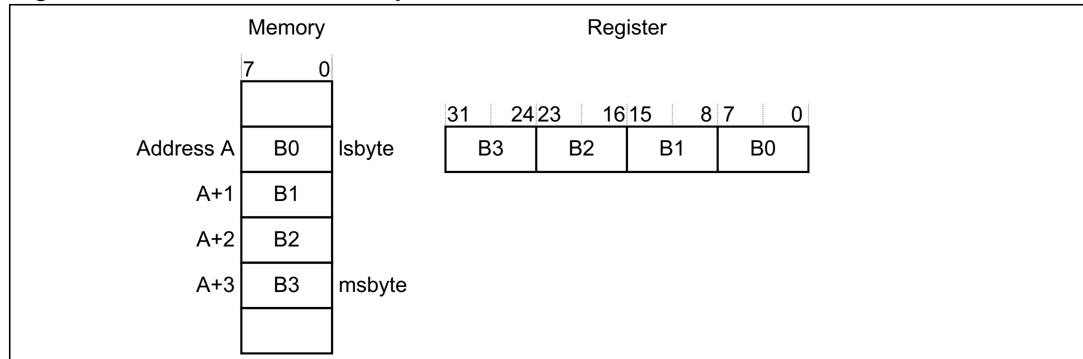
2.2.5 Memory endianness

The processor views memory as a linear collection of bytes numbered in ascending order from zero. For example, bytes 0-3 hold the first stored word, and bytes 4-7 hold the second stored word.

Little-endian format

In little-endian format, the processor stores the least significant byte (lsbyte) of a word at the lowest-numbered byte, and the most significant byte (msbyte) at the highest-numbered byte. See *Figure 7* for an example.

Figure 7. Little-endian example



2.3 Exception model

This section describes the exception model.

2.3.1 Exception states

Each exception is in one of the following states:

Inactive: The exception is not active and not pending.

Pending: The exception is waiting to be serviced by the processor. An interrupt request from a peripheral or from software can change the state of the corresponding interrupt to pending.

Active: An exception that is being serviced by the processor but has not completed.

Note: An exception handler can interrupt the execution of another exception handler. In this case both exceptions are in the active state.

Active and pending: The exception is being serviced by the processor and there is a pending exception from the same source.

2.3.2 Exception types

The exception types are:

Reset: Reset is invoked on power up or warm reset. The exception model treats reset as a special form of exception. When reset is asserted, the operation of the processor stops, potentially at any point in an instruction. When reset is deasserted, execution restarts in Thread mode from the address provided by the reset entry in the vector table.

NMI: A *NonMaskable Interrupt* (NMI) can be signalled by a peripheral or triggered by software. This is the highest priority exception other than reset. It is permanently enabled and has a fixed priority of -2. NMIs cannot be:

- Masked or prevented from activation by any other exception
- Preempted by any exception other than Reset.

Hard fault: A hard fault is an exception that occurs because of an error during normal exception processing. Hard faults have a fixed priority of -1, meaning they have higher priority than any exception with configurable priority.

SVCall: A *supervisor call* (SVC) is an exception that is triggered by the SVC instruction. In an OS environment, applications can use SVC instructions to access OS kernel functions and device drivers.

PendSV: PendSV is an interrupt-driven request for system-level service. In an OS environment, use PendSV for context switching when no other exception is active.

SysTick: A SysTick exception is an exception the system timer generates when it reaches zero. Software can also generate a SysTick exception. In an OS environment, the processor can use this exception as system tick.

Interrupt (IRQ) A interrupt, or IRQ, is an exception signalled by a peripheral, or generated by a software request. All interrupts are asynchronous to instruction execution. In the system, peripherals use interrupts to communicate with the processor.

Table 12. Properties of the different exception types

Exception number[1]	IRQ number[1]	Exception type	Priority	Vector address or offset[2]	Activation
1	-	Reset	-3, the highest	0x00000004	Asynchronous
2	-14	NMI	-2	0x00000008	Asynchronous
3	-13	Hard fault	-1	0x0000000C	Synchronous
4-10	-	Reserved	-	-	-
11	-5	SVCall	Configurable[3]	0x0000002C	Synchronous
12-13	-	Reserved	-	-	-
14	-2	PendSV	Configurable[3]	0x00000038	Asynchronous
15	-1	SysTick	Configurable[3]	0x0000003C	Asynchronous
16 - 47	0 - 31	Interrupt (IRQ)	Configurable[3]	0x00000040 and above[4]	Asynchronous

1. To simplify the software layer, the CMSIS only uses IRQ numbers and therefore uses negative values for exceptions other than interrupts. The IPSR returns the Exception number, see *Interrupt program status register on page 14*.
2. See *Vector table on page 24* for more information.
3. See *Interrupt priority register (IPR0-IPR7) on page 73*.
4. Increasing in steps of 4.

For an asynchronous exception other than reset, the processor can execute another instruction between when the exception is triggered and when the processor enters the exception handler.

Software can disable the exceptions that *Table 12 on page 23* shows as having configurable priority, see:*Interrupt clear-enable register (ICER) on page 71*.

For more information about hard faults, see *Section 2.4: Fault handling on page 28*.

2.3.3 Exception handlers

The processor handles exceptions using:

Interrupt Service Routines (ISRs) Interrupts IRQ0 to IRQ31 are the exceptions handled by ISRs.

Fault handlers Hard fault is the only fault exception handled by the fault handlers.

System handlers NMI, PendSV, SVCall SysTick, and Hard fault exceptions are all system exceptions that are handled by system handlers.

2.3.4 Vector table

The vector table contains the reset value of the stack pointer, and the start addresses, also called exception vectors, for all exception handlers.

Figure 8 shows the order of the exception vectors in the vector table.

The least-significant bit of each vector must be 1, indicating that the exception handler is Thumb code.

Figure 8. Vector table

Exception number	IRQ number	Vector	Offset
47	31	IRQ31	0xBC
. . .		. . .	. . .
18	2	IRQ2	0x48
17	1	IRQ1	0x44
16	0	IRQ0	0x40
15	-1	SysTick	0x3C
14	-2	PendSV	0x38
13		Reserved	
12			
11	-5	SVCall	0x2C
10		Reserved	
9			
8			
7			
6			
5			
4			0x10
3	-13	HardFault	0x0C
2	-14	NMI	0x08
1		Reset	0x04
		Initial SP value	0x00

MS30030V1

On system reset, the vector table is fixed at address 0x00000000.

2.3.5 Exception priorities

As *Table 12 on page 23* shows, all exceptions have an associated priority, with:

- A lower priority value indicating a higher priority
- Configurable priorities for all exceptions except Reset, Hard fault, and NMI.

If software does not configure any priorities, then all exceptions with a configurable priority have a priority of 0. For information about configuring exception priorities see:

- *System handler priority registers (SHPRx) on page 83*
- *Interrupt priority register (IPR0-IPR7) on page 73*

Configurable priority values are in the range 0-192, in steps of 64. This means that the Reset, Hard fault, and NMI exceptions, with fixed negative priority values, always have higher priority than any other exception.

For example, assigning a higher priority value to IRQ[0] and a lower priority value to IRQ[1] means that IRQ[1] has higher priority than IRQ[0]. If both IRQ[1] and IRQ[0] are asserted, IRQ[1] is processed before IRQ[0].

If multiple pending exceptions have the same priority, the pending exception with the lowest exception number takes precedence. For example, if both IRQ[0] and IRQ[1] are pending and have the same priority, then IRQ[0] is processed before IRQ[1].

When the processor is executing an exception handler, the exception handler is preempted if a higher priority exception occurs. If an exception occurs with the same priority as the exception being handled, the handler is not preempted, irrespective of the exception number. However, the status of the new interrupt changes to pending.

2.3.6 Exception entry and return

Descriptions of exception handling use the following terms:

Preemption — When the processor is executing an exception handler, an exception can preempt the exception handler if its priority is higher than the priority of the exception being handled. When one exception preempts another, the exceptions are called nested exceptions. See *Exception entry on page 26* more information.

Return — This occurs when the exception handler is completed, and:

- There is no pending exception with sufficient priority to be serviced
- The completed exception handler was not handling a late-arriving exception.

The processor pops the stack and restores the processor state to the state it had before the interrupt occurred. See *Exception return on page 27* for more information.

Tail-chaining — This mechanism speeds up exception servicing. On completion of an exception handler, if there is a pending exception that meets the requirements for exception entry, the stack pop is skipped and control transfers to the new exception handler.